A Finite-Precision Adaptation of Bit Recycling to Arithmetic Coding

Ahmad Al-Rababa'a
Université Laval, Canada
Ahmad.Al-Rababaa.1@ulaval.ca

Danny Dubé
Université Laval, Canada
Danny.Dube@ift.ulaval.ca

***Abstract*—The bit recycling compression technique has been introduced to minimize the redundancy caused by the multiplicity of encodings present in many compression techniques. It has achieved about 9% as a reduction in the size of the files compressed by Gzip. In prior work, we have proposed an arbitrary-precision technique to adapt bit recycling to arithmetic code instead of Huffman code. We have shown that this adaptation enables bit recycling to achieve better compression and a much wider applicability. We have also presented a theoretical analysis that estimates the average amount of data compression that can be achieved by this adaptation. In this paper, we propose the finite-precision version of this adaptation so that it can be implemented efficiently using conventional computer registers.**

I. INTRODUCTION

Data compression aims to reduce the size of data so that it requires less storage space and less bandwidth of the communication channels. Data compression has recently started to be used to reduce the energy consumption in many wireless applications, such as wireless-networked handheld devices [1] and wireless sensor networks [2], since wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation [3]. Many compression techniques suffer from a problem that we call the *redundancy caused by the multiplicity of encodings* (ME). ME means that the source data may be encoded in more than one way. In its simplest form, it occurs when a compression technique with ME has the freedom, at certain steps during the encoding process, to encode the next symbol in different ways; i.e. different codewords for the same symbol can be sent to the decoder and any one of theses codewords can be decoded correctly. Upon occurrence of such a situation, the default behavior of most techniques is to encode the symbol using the shortest codeword and, possibly, the least computation. Many applications suffer from ME, such as LZ77 (Lempel and Ziv, 1977) and its variants, some variants of the Prediction by Partial Matching (PPM) technique, Volf and Willems switching compression technique [4], and Knuth's algorithm [5] for the generation of balanced codes.

The *Bit Recycling* (BR) technique has been introduced to reduce the redundancy caused by ME [6]. It reduces that kind of redundancy by harnessing ME in a certain way, so that it is not always necessary to select the shortest codeword, but instead, so that all the appropriate codewords are taken into account with some agreement between the encoder and the decoder. Variants of BR have been applied on LZ77 algorithm by Dubé and Beaudoin. The experimental results showed that BR has achieved better compression (a reduction of about 9% in the size of files that has been compressed by Gzip) by exploiting ME rather than systematically selecting the shortest codeword [7], [8].

The authors of BR have pointed out that their technique could not minimize the redundancy perfectly since it is built on Huffman codes (HC), which does not have the ability to deal with codewords of fractional lengths; i.e. it is constrained to generate codewords of integral lengths. Moreover, Huffman-Coding-based BR (HCBR) has imposed additional burdens to avoid some situations that affect its performance negatively. Unlike HC, Arithmetic Coding (AC) does have the capability of manipulating codewords of fractional lengths. Furthermore, it has attracted the researchers in the last few decades since it is more powerful and flexible than HC. Consequently, a new technique named *Arithmetic-Coding-based BR* (ACBR) has been proposed to resolve the weakness of HCBR by adapting it to AC [9]. A theoretical analysis showed that ACBR achieves perfect recycling in all cases whereas HCBR achieves perfect recycling only in very specific cases. Accordingly, significantly better compression could be obtained using ACBR.

The problem of ACBR, as proposed by the authors, is that it uses *arbitrary-precision* calculations, which require unbounded (or infinite) resources [9]. Hence, in order to benefit from ACBR in practice, ACBR needs to be adapted so that it can perform *finite-precision* calculations instead of arbitrary-precision calculations. This would make it efficiently applicable on computers with conventional fixed-size registers. This work aims to address the problems of arbitrary-precision ACBR (APACBR).

The outline of the next sections are as follows. In Section II, we briefly review the LZ77 technique, ME, the objectives of HCBR and its weakness, the principle of APACBR, and the problems of APACBR. In Section III, we present a finite-precision technique that addresses the problems of APACBR. The new proposed technique consists of two algorithms: the coder and the decoder. Finally, the conclusion and future work are given in Section IV.

II. BACKGROUND

A. The principle of LZ77, HCBR, and ME

The reasons of the remaining redundancy in compressed data are: inappropriate modeling of the data, incorrect (or inaccurate) random source statistics, and ME. The concern of this work is the third reason: ME. Let us show an instance of redundancy caused by ME by using the following LZ77 example. LZ77 is a compression technique that compresses a

TABLE I. THE PROBABILITY DISTRIBUTION AND THE CORRESPONDING HUFFMAN CODEWORDS OF α AT TIME t.

Message (m_i)	Count (Cnt_i)	Probability (p_i)	Cumulative probability (Q_i)	Huffman codeword (Huff_i)
m_0	3	0.064	0.000	0101
m_1 (M_1)	15	0.319	0.064	11
m_2	12	0.255	0.383	10
m_3 (M_2)	3	0.064	0.638	0100
m_4	8	0.170	0.702	00
m_5 (M_3)	6	0.128	0.872	011
Total	47	1.000		

string of characters, S, by transmitting a sequence of messages. A message is either a *literal* message, denoted by $[c]$, which means that the next character is c, or a *match* message, denoted by $\langle l, d\rangle$, which means that the next l characters are identical to those located at distance d prior to the current position in S. For example, let S be "$\underline{abbaaabbabbabb}b$", the underlined substring is the prefix that has been encoded so far. The next character to be encoded is "a", which can be encoded by transmitting the literal message $[a]$. However, the encoder can also encode, for instance, "*abb*" by transmitting any one of the match messages $\langle 3,3\rangle$, $\langle 3,6\rangle$, and $\langle 3,11\rangle$, since "*abb*" has three copies at the distances 3, 6, and 11 in the underlined substring. LZ77 typically selects the *longest* match ("*abb*") at the *closest* distance ($d = 3$), therefore the match message $\langle 3,3\rangle$ will be transmitted and the encoder proceeds to the last "*b*". It is clear that LZ77 has the *freedom* to encode "*abb*" by selecting any message from *the set of the equivalent messages* $\mathcal{M}_\equiv = \{M_1, M_2, M_3\}$, where $M_1 = \langle 3,3\rangle$, $M_2 = \langle 3,6\rangle$, and $M_3 = \langle 3,11\rangle$. These messages are called the equivalent messages since any message in $\mathcal{M}_\equiv$ can be used to encode "*abb*". Accordingly, many different sequences of messages may be transmitted to describe S, and any possible sequence will be decoded correctly. It is clear that this property represents an instance of ME.

BR aims at improving code efficiency by exploiting the redundancy caused by ME [6]. In BR, the compressor is not restricted to select the default choice, i.e. the shortest codeword, and to ignore the other choices. Instead, thanks to some *agreement* between the compressor and decompressor, it *uses* ME to *implicitly* carry information from the compressor to the decompressor. Let us illustrate this using the following example, which will be used as a running example in the remainder of this paper. Assume that, at time t, the alphabet α is $\{m_i\}_{i=0}^{5}$ and the corresponding distribution and Huffman codewords for α are as shown in Table I. Let $\mathcal{M}_\equiv = \{M_1, M_2, M_3\}$ be the set of *equivalent* messages at time t, where $M_1 = m_1$, $M_2 = m_3$, and $M_3 = m_5$ (similar to the equivalent messages for "*abb*" in the above example). Suppose that the string to be encoded, S, can be described first using any message in $\mathcal{M}_\equiv$ at time t, and then, at time $t+1$, using solely message m_3 (without equivalents). The Huffman codeword Huff_i corresponding to message m_i is generated using HC, based on the count of occurrences (Cnt_i) of m_i. The default behavior (without recycling) is that the equivalent message with the shortest codeword, $M_1 = m_1$, gets selected and Huff_1 (i.e. 11) is transmitted to the decoder.

Let us review the principle of HCBR and show how it can reduce the default cost ($C_{\mathrm{default}} = 2$ bits) by adding some computations and the exchange of extra information between the model and coder/decoder. At time t, the compressor has the freedom to encode M_1, M_2, or M_3 of costs 2 bits, 4 bits, and 3 bits, respectively. The compressor, using HC, constructs a *prefix codeword* r_i for each M_i according to the corresponding Cnt_i; let us say 0, 10, and 11 for M_1, M_2, and M_3, respectively. The created codewords, are called the *recycled codewords*. Thus, each M_i will have two codewords: the *explicit* codeword Huff_i that will be sent to the decoder, and the corresponding *implicit* codeword r_i that will be constructed implicitly and identically by the compressor and the decompressor as follows. Suppose that each r_i is compared with the first bit(s), let us say b_0b_1, of the codeword of the next message, then definitely, one and only one match should occur. The equivalent message whose r_i matches b_0b_1 is sent to the decompressor. In other words, b_0 should be either 0 or 1. If $b_0 = 0$, select M_1. Otherwise, b_0b_1 has to be either 10 or 11. If $b_0b_1 = 10$, select M_2. Otherwise, select M_3. The decompressor, after receiving one particular M_i, $i \in \{1, 2, 3\}$, would be able to acknowledge that the selection of the received message among M_1, M_2, and M_3 was intentional and it would then be able to recover r_i from the received M_i. Accordingly, the compressor is freed from the obligation to insert the matched bits into the compressed stream since the decompressor can implicitly deduce the corresponding r_i from the received Huff_i and restore them at the same location.

Let us evaluate the performance of HCBR in the previous example. The average net cost NC of the set of equivalent messages, $\mathcal{M}_\equiv = \{M_i\}_{i=1}^{n}$, is given by:

$$\mathrm{NC} = \sum_{i=1}^{n} (c_i - |r_i|) \times \frac{1}{2^{|r_i|}}, \tag{1}$$

where n is the number of the equivalent messages, c_i is the cost of M_i, i.e. the length of its codeword ($c_i = |\mathrm{Huff}_i|$), and $|r_i|$ is the length of the recycled codeword. So NC for $\mathcal{M}_\equiv$ is 1.25 bits, which is less than the default cost ($C_{\mathrm{default}} = 2$ bits). This benefit is brought by HCBR but did BR achieve the perfect (maximum) recycling by this average net cost? To answer this question, we first need to know what is the *ideal* (minimum) average net cost of $\mathcal{M}_\equiv$ according to the associated probabilities. The self-information of $\mathcal{M}_\equiv$, say T, represents the minimum average net cost of $\mathcal{M}_\equiv$, and T is given by:

$$T = -\log \sum_{i=1}^{n} p_i. \tag{2}$$

The value of T in our example is 0.97 bit, so HCBR did not achieve perfect recycling. The reason behind this is that HC is constrained to generate recycled codewords of integer lengths, which correspond to probabilities that are powers of $\frac{1}{2}$ only. Therefore, HCBR could not achieve perfect recycling due to the nature of HC. Moreover, HCBR has imposed the additional burden to avoid some situations that affect its performance negatively. AC is more flexible and it does have the ability to utilize the ratio between the messages' probabilities fractionally and to recycle fractions of bits. Accordingly, the authors have proposed APACBR to resolve this weakness and to improve the code efficiency and the flexibility of BR [9]. Next, we explain the principle of APACBR.

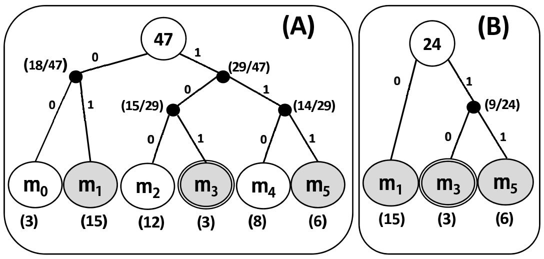

Fig. 1. (A) The binary tree of α at time t. (B) The skeleton tree of the equivalent messages.

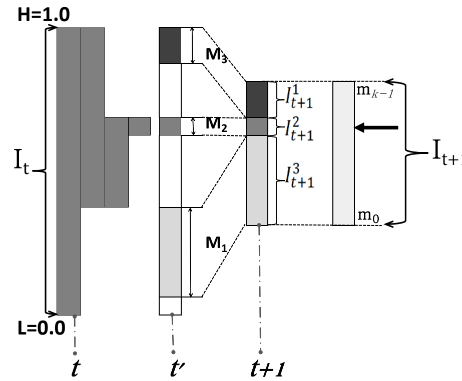

Fig. 2. The principle of APACBR.

B. The principle of APACBR

We consider the same running example to describe the principle of APACBR. Let us first describe the model that will be used in our technique. The alphabet shown in Table I can be decomposed into the corresponding binary tree depicted in Figure 1, which represents the statistical model at time t. This decomposition enables the model to accommodate a larger and more skewed alphabets. The root node in Figure 1-A, contains the total counts (47) of the alphabet messages, the leaves of the tree represent α. The shaded leaves represent $\mathcal{M}_{\equiv}$ at time t. The skeleton tree depicted in Figure 1-B is derived from the main tree and describes the relative weights and locations of equivalent messages. The trees in Figure 1-A and Figure 1-B will be used by the model to provide the encoder/decoder with necessary information to perform coding/decoding and recycling, respectively.

The statistical model describes the message to be encoded by transmitting a sequence of binary events B associated with the corresponding probability, P_0 or P_1 (we assume $P_0 + P_1 = 1$), that are formed by traversing from the root to the leaf that corresponds to the message to be encoded. For example, to encode $M_2 = m_3$ as one of the equivalent messages at time t, the model has to send the following sequence of coding orders of the form $(B,\, P_0,\, P_1)$ to the encoder: $(1,\, \frac{18}{47},\, \frac{29}{47})$, $(0,\, \frac{15}{29},\, \frac{14}{29})$, and $(1,\, \frac{12}{15},\, \frac{3}{15})$. Since ACBR uses AC to encode S, the encoder starts executing each order by gradually dividing the unit interval $[0,\, 1)$ into two subintervals according to P_0 and P_1. The interval of event 0 is I_{Evt_0} and that of event 1 is I_{Evt_1}. Only one subinterval is kept, from time t to time t', according to B, as shown in Figure 2. Similarly, the skeleton tree is used to transmit a sequence of recycling orders to perform recycling as we will show later in this paper.

Let the subinterval corresponding to the equivalent message M_i at time t, denoted by I^i_t. APACBR aims to let any M_i from $\mathcal{M}_{\equiv}$ be selected according to the next message, providing that the next message, at $t+1$, will be encoded using the total of $\mathcal{M}_{\equiv}$ sub-intervals I_{t+1} instead of the subinterval of the selected message I^i_{t+1} as shown in Figure 2, which results in fewer bits to be sent to the decoder to describe any message at $t+1$. The length of the interval at time $t+1$ represents the self-information of the available equivalent messages; i.e. $\#I_{t+1} = \#I^1_{t+1} + \#I^2_{t+1} + \#I^3_{t+1}$.

Let us assume that the location (cumulative probability) of the messages at time $t+1$ is indicated by the *arrow* in Figure 2. According to the location of the arrow, one and only one of the equivalent messages will be selected (and encoded) as follows. Since the arrow points to I_{t+1} and to I^2_{t+1} at the same time and I^2_{t+1} belongs to M_2 (in its scope), then the encoder selects message M_2. The encoder provides the new interval I_{t+1} for the next message to be encoded instead of I^2_{t+1}. The same thing for M_1 and M_3 as follows. If the arrow points above I^2_{t+1}, so it points to I^3_{t+1}, which is the part of I_{t+1} related to M_3, accordingly, the encoder selects M_3 and so on. Hence, each M_i in $\mathcal{M}_{\equiv}$ has an opportunity to be encoded using I_{t+1} instead of I^i_{t+1} but one and only one of them will be used according to the cumulative probability of next message to be encoded. Widening the interval from I^2_{t+1} to I_{t+1} represents the *arithmetic recycling* according to equation (2).

On the other side, the decoder undoes what the encoder did based on the aforementioned arrangement as follows. The decoder at time t will have the same information represented by the tree in Figure 1 based on the already decoded string (before t). The decoder starts decoding using the value represented by the position of the arrow with respect to I_t. The model at this point has to provide the decoder with P_0 $(\frac{18}{47})$ so that the decoder can accordingly split I_t into I_{Evt_0} and I_{Evt_1}, now the decoder can determine that the arrow is pointing to the upper part, i.e. I_{Evt_1}, accordingly the decoder tells the model that the first decoded binary event B is 1, which tells the model that the message to be encoded is located to the right in the tree in Figure 1. The decoder continues using the same procedure until the model reaches the leaf (m_3) at time t', where the decoder can realize that the decoded message $(M_2 = m_3)$ has two other equivalents, M_1 and M_3 (recall LZ77), accordingly, it has to rebuild I_{t+1} based on the decoder's implicit knowledge about $\mathcal{M}_{\equiv}$ represented by the skeleton tree Figure 1-B; i.e. the decoder can retrieve the necessary information that enables the decoder to decode the next message according to I_{t+1} instead of I^2_{t+1}. At time $t+1$, the same position of the arrow with respect to I_{t+1} will be used to decode the next message as described above and so on.

The authors have compared the performance of ACBR and HCBR in terms of the compression efficiency and time complexity [9]. We found that *the average net cost*, NC, of any message in $\mathcal{M}_{\equiv}$, that can be achieved by HCBR, NC_H, is bounded by $T \leq \mathrm{NC}_H \leq C_{\mathrm{default}}$, and $\mathrm{NC}_H \approx T$ only when the messages in $\mathcal{M}_{\equiv}$ have equal probabilities, while NC of ACBR, $\mathrm{NC}_A \approx T$ in all cases, regardless of the ratio between the probabilities of the messages in $\mathcal{M}_{\equiv}$.

Furthermore, in recent work, the authors have used both HCBR and APACBR as the means to reduce the redundancy caused by ME in plurally parsable dictionaries designed by Savari [10], [11]. We were able to reduce this redundancy significantly, and we have theoretically shown that ACBR (in general) is more efficient and flexible than HCBR, but due to some shortcomings in the applicability of APACBR, we could not evaluate its performance in practice. However, we next discuss these shortcomings in detail.

C. The problems and the associated needs of APACBR

The main problem of the technique described above is that it uses arbitrary-precision AC (APAC) to encode each message at a time, which entails too many resources and makes it impractical, therefore we need to adapt the encoder to finite-precision AC (FPAC) so that we can reduce the computational requirements and it can then be applied in practice. Next, we discuss (using the same example) the basic needs, settings, and consequences that are required to address this problem.

Traditional FPAC encodes one message, m_i, at a time, as we have shown above, the ACBR coder has to be able to encode and keep track of more than one message ($\mathcal{M}_{\equiv}$) at a time, so that the ACBR coder can exploit the self information of $\mathcal{M}_{\equiv}$, and consequently, achieve more compression. Hence, the ACBR coder has to proceed *non-deterministically* (ND) by assigning a separate thread, Θ_i for each $M_i \in \mathcal{M}_{\equiv}$.

In FPAC, the unit interval $[0,\,1)$ gets mapped into $[0,\,N) = \big[0,\,2^b\big)$, where AC uses b-bit registers to encode S, $[L,\,H)$ is defined as $\{x \mid L \le x < H\}$. The interval $[L,\,H)$ is updated for each encoding step, when the width of the interval shrinks to a certain limit, the encoder according to the values of L and H, performs one of the three types of *upscaling* as follows: E1, E2 or E3 when $[L,\,H)$ lies in $\big[0,\,\frac{N}{2}\big)$, $\big[\frac{N}{2},\,N\big)$, or $\big[\frac{N}{4},\,\frac{3N}{4}\big)$ respectively. According to FPAC, the encoder at time t', in the above example, needs first to *upscale* (enlarge) $I^2_{t'}$ so that it becomes large enough to encode the message at time $t+1$. At time $t+1$, the encoder needs to merge the intervals of the other equivalent messages, but it has to consider the new length, $\#I^2_{t+1}$, as a reference after upscaling. As we assumed above that $\#I^1_t = 5 \times \#I^2_t$ and $\#I^3_t = 2 \times \#I^2_t$, then the total length of merged interval at time $t+1$ would exceed the used finite-precision limits. To address this raised problem, we need to *downscale* the interval so that we can accommodate the intervals of the other equivalent messages within the used finite limits. Hence, we propose next a finite-precision variant of ACBR (FPACBR) that addresses the aforementioned problem and the associated needs.

III. Description of FPACBR

In this section, we propose FPACBR, which has the following main features. It has to be based on AC and to proceed ND for the reasons explained above, and it can be easily interfaced with the models of several different compression techniques that suffer from the redundancy caused by ME. We use the same running example to explain the pseudo-code shown in Figure 3 for the encoder algorithm, on lines 1–63, and the decoder algorithm, on lines 65–84, with the help of Figure 4, which illustrates the steps of encoding $M_2 = m_3$

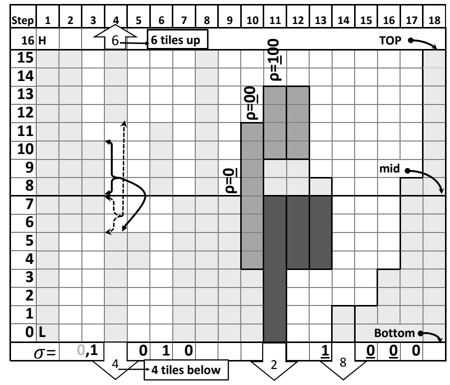

Fig. 4. The steps of encoding $\{M_1, \underline{M_2}, M_3\}$ m_3, where $M_1 = m_1$, $M_2 = m_3$, and $M_3 = m_5$.

at time t followed by m_3 at time $t+1$ based on a sequence of *encoding* and *recycling* orders received from the model. The encoder interprets the received ordered into a compressed binary stream σ that will be sent to decoder.

In APACBR, the limits of the intervals and the arrow were regarded as *points* (real numbers) in the unit interval $[0,\,1)$. Here, we need to map these points into *tiles*, and the arrow into pointed tile. The FPACBR coder uses a convention different from that used by FPAC, which will be explained throughout this section.

Let $\#I$ be the number of tiles occupied by an interval I. $\#I$ is now an integer. The coder uses a *global* interval G of at most 2^b tiles, and inside G, the encoder uses a *fixed* central interval, $F = [Bot,\,Top)$ of length $\#F = (Top - Bot) = \mathbf{B} = 2^{b-2}$ tiles. The *working* interval $W = [L,\,H)$ is initialized to $[Bot,\,Top)$. The size of W increases and decreases according to the coding steps and it is bounded by $1 = \mathbb{B} \le \#W \le \mathbf{B}$. L and H are allowed to slide outside of F for a limited number of tiles, $\mathbf{B} - 1$. The interval F contains the *active interval*, $A = [a_1,\,a_2) \subseteq W$, which is the portion of W that belongs to the current thread Θ_{cur}. The ratio of A to W in Θ_{cur} represent the relative weight and location of M_i to the weights and locations of $\mathcal{M}_{\equiv}$. Accordingly, the *inactive (rest of) interval* will be $I_{\text{inact}} = W - A$, which is the portion of W that belongs to other concurrent threads. Note that I_{inact} need not be an interval.

To encode the first message ($M_2 = m_3$) at time t, the model sends the same sequence of coding orders that has been sent to the APAC encoder: $(1,\,\frac{18}{47})$, $(0,\,\frac{15}{29})$, and $(1,\,\frac{12}{15})$. The encoder and decoder follow the same principle described in APACBR, but using a finite-precision coding/decoding. Thus, the encoder gradually divides the initial interval $I_{init} = [L,\,H) = [0,\,16)$ into two subintervals, I_{Evt_0} and I_{Evt_1}, of integer lengths according to P_0 using the rounding rule stated in line 2 in the pseudo-code, the encoder then considers either I_{Evt_0} or I_{Evt_1} according to B to continue executing the next coding orders as stated on lines 3–7 in the pseudo-code. The steps of encoding the aforementioned sequence of coding orders are illustrated in Figure 4, from steps 1 to 5. The decoder uses the same procedure followed by the encoder to split the current interval according to P_0, and according to the arrow location, it decodes the associated binary event. The

1: **procedure Encode**(B, P_0, P_1)
2: $x \leftarrow \lfloor(\#W \times P_0) + \frac{1}{2}\rfloor$
3: $x \leftarrow \min(\max(1,\, x),\, \#W - 1)$
4: **if** $B = 0$ **then**
5: $\quad W \leftarrow [L,\, L + x)$
6: **else**
7: $\quad W \leftarrow [L + x,\, H)$
8: **if** $W \cap A = \emptyset$ **then**
9: $\quad$ **abort** Θ_{cur} /* current thread */
10: **trim** A **to** W
11: **while** $\#W \leq \mathit{Half}$ **do**
12: $\quad$ **if** $A \subseteq [\mathit{Bot},\, \mathit{Mid})$ **then**
13: $\quad\quad W \leftarrow \mathbf{E1}(W)$
14: $\quad\quad A \leftarrow \mathbf{E1}(A)$
15: $\quad\quad w \leftarrow w \cdot 0$
16: $\quad$ **else if** $A \subseteq [\mathit{Mid},\, \mathit{Top})$ **then**
17: $\quad\quad W \leftarrow \mathbf{E2}(W)$
18: $\quad\quad A \leftarrow \mathbf{E2}(A)$
19: $\quad\quad w \leftarrow w \cdot 1$
20: $\quad$ **else**
21: $\quad\quad$ **fork** Θ_{cur} **into** Θ_0, Θ_1:
22: $\quad\quad$ In Θ_0: **trim** A **to** $[L,\, \mathit{Mid})$
23: $\quad\quad$ In Θ_1: **trim** A **to** $[\mathit{Mid},\, H)$
24: **Emit**(w)
25:
26: **procedure E1**($[l,\, h)$):
27: **return** $[2 \times l - \mathit{Bot},\, 2 \times h - \mathit{Bot})$
28:
29: **procedure E2**($[l,\, h)$):
30: **return** $[2 \times l - \mathit{Top},\, 2 \times h - \mathit{Top})$
31: **procedure Emit**(w):
32: **if** $w = \epsilon \vee \rho = \epsilon$ **then**
33: $\quad \sigma \leftarrow \sigma \cdot w$
34: **else if** $\text{head}(w) = \text{head}(\rho)$ **then**
35: $\quad \rho \leftarrow \text{tail}(\rho)$
36: $\quad$ **Emit**(tail(w))
37: **else**
38: $\quad$ **abort** Θ_{cur} /* current thread */
39:
40: **procedure RecycleE**(R, P_0, P_1)
41: $P_1 \leftarrow 1 - P_0$
42: $P \leftarrow P_{1-R}$ /* prob. of Θ_{cur}'s option */
43: $\mathit{Ext} \leftarrow \left\lfloor \#W \times (\frac{1}{P} - 1) \right\rfloor$
44: **while** $\#W > \max(\mathbf{B} - \mathit{Ext}, \mathrm{B})$ **do**
45: $\quad$ **if** $(R = 0)$ **then**
46: $\quad\quad W \leftarrow \mathbf{S2}(W)$
47: $\quad\quad A \leftarrow \mathbf{S2}(A)$
48: $\quad\quad \rho \leftarrow 1 \cdot \rho$
49: $\quad$ **else**
50: $\quad\quad W \leftarrow \mathbf{S1}(W)$
51: $\quad\quad A \leftarrow \mathbf{S1}(A)$
52: $\quad\quad \rho \leftarrow 0 \cdot \rho$
53: $\quad \mathit{Ext} \leftarrow \left\lfloor \#W \times (\frac{1}{P} - 1) \right\rfloor$
54: $\mathit{Ext} \leftarrow \min(\mathit{Ext},\, \mathbf{B} - \#W)$
55: **if** $(R = 0)$ **then**
56: $\quad W \leftarrow [L - \mathit{Ext},\, H)$
57: **else**
58: $\quad W \leftarrow [L,\, H + \mathit{Ext})$
59: **procedure S1**($[l,\, h)$)
60: **return** $\left[\left\lfloor \frac{\mathit{Bot}+l}{2} \right\rfloor,\, \left\lceil \frac{\mathit{Bot}+h}{2} \right\rceil\right)$
61:
62: **procedure S2**($[l,\, h)$)
63: **return** $\left[\left\lfloor \frac{l+\mathit{Top}}{2} \right\rfloor,\, \left\lceil \frac{h+\mathit{Top}}{2} \right\rceil\right)$
64:
65: **procedure Decode**(P_0, P_1)
66: Calculate x as in lines 2–3
67: **if** $v < L + x$ **then**
68: $\quad \mathit{Evt} \leftarrow 0$ /* Evt_0 gets decoded */
69: $\quad W \leftarrow [L,\, L + x)$
70: **else**
71: $\quad \mathit{Evt} \leftarrow 1$ /* Evt_1 gets decoded */
72: $\quad W \leftarrow [L + x,\, H)$
73: **while** $\#W \leq \mathit{Half}$ **do**
74: $\quad$ **if** $\mathit{MSB}(v) = 0$ **then**
75: $\quad\quad W \leftarrow \mathbf{E1}(W)$
76: $\quad$ **else**
77: $\quad\quad W \leftarrow \mathbf{E2}(W)$
78: $\quad$ Shift v to the left by one bit
79: $\quad$ Read σ's next bit into $\mathit{LSB}(v)$
80: **return** Evt
81:
82: **procedure RecycleD**(R, P_0, P_1)
83: /* Like **RecycleE** but replace */
84: /* ρ by v in lines 48 and 52. */

Fig. 3. Pseudo-code for the algorithms of the encoder and the decoder.

arrow location here is represented by a finite number $(b-2)$ of bits of the compressed stream σ as we will show next. Let us show this by executing the first coding order $(1,\ \frac{18}{47})$ at time t. Initially, at time t, $\#I_{init} = 16$, and $P_0 = \frac{18}{47}$, so $\#I_{Evt_0} = \#I_{init} \times P_0 = 6.13$, the result (6.13) gets rounded to 6, and therefore $\#I_{Evt_1} = 16 - 6 = 10$. According to $B = 1$, the encoder ignores I_{Evt_0} and considers I_{Evt_1} as the current interval to execute the next orders. Thereby W is updated to $[L,\, H) = [6,\, 16)$.

During the encoding/decoding process, if W shrinks to less than or equal to *half* the length of F, that is $\mathit{Half} = \frac{\mathit{Top}-\mathit{Bot}}{2}$, the coder is triggered to upscale W so that it is scaled up above Half tiles as stated on lines 11–23 in the pseudo-code, the encoder can then continue executing the next orders. The coder performs only two types of upscaling, E1 and E2 according to the location of A and the mid point of F, $\mathit{Mid} = \frac{\mathit{Top}+\mathit{Bot}}{2}$, as follows. E1 is performed if $A \subseteq [\mathit{Bot},\, \mathit{Mid})$, yielding 0 to σ. E2 is performed if $A \subseteq [\mathit{Mid},\, \mathit{Bot})$, yielding 1 to σ. Such yielded bits (1010), indicated at the lowest row in Figure 4, are kept temporarily in register w. Since the coder proceeds ND, we chose to avoid E3 upscaling completely. When A straddles Mid, instead of using E3, we take advantage of the ND process, as shown at steps 3 and 4 in Figure 4 and as stated on lines 21–23 in the pseudo-code. This is achieved by splitting Θ_{cur} into two new separate threads, Θ_0 and Θ_1, each with their specific active interval. The encoder will enventually kill the non-proper thread according to the upcoming coding orders.

The decoder uses the same two types of upscaling, E1 and E2, but in different way as described in the pseudo-code on lines 73–79 and as follows. Initially, the decoder loads register v of size $b-2$ (i.e. 4) bits with the first 4 bits of σ ($v = 1010$). The decoder performs E1 or E2 upscaling according to the Most Significant Bit of v, $\mathit{MSB}(v)$. For each upscaling, the contents of v is shifted left by one bit and one bit is consumed from σ to become the Least Significant Bit of v, $\mathit{LSB}(v)$.

The situations at step 8 in Figure 4, represent the greatest challenge of FPACBR, since the encoder needs to merge the corresponding intervals of the other equivalent messages to the current interval $W = [0,\, 16)$ after upscaling, and the whole merged interval should be accommodated within the limits of G. To do so, another procedure named **RecycleE** is required. The main purpose of **RecycleE** is to downscale W until it becomes small enough to accommodate the whole merged interval within G as described in the pseudo-code on lines 40–58. Therefore, the model has to provide **RecycleE** with a sequence of *recycling orders* that describe the relative weights and locations of $\mathcal{M}_{\equiv}$ using the *skeleton* tree of $\mathcal{M}_{\equiv}$ shown in Figure 1-B as follows. Let us describe the recycling orders using our example. Starting from the internal node labeled $\frac{9}{24}$ (9/24 in the picture), which connects the current message m_3 with the *first* (closest) neighbor m_5, the position of the first neighbor (m_5) is to the right, let us say $R = 1$, of the internal node labeled $\frac{9}{24}$, and the weight of the left branch is $P_0 = \frac{3}{9}$. In other words, R tells the encoder/decoder if the interval to be merged is above ($R = 1$) or below ($R = 0$) of the current interval, and the associated P_0 tells the encoder/decoder about the ratio of the current interval to the interval to be merged. So the recycling order $(1,\, \frac{3}{9})$ of the form $(R,\, P_0)$ describes m_5 as the first equivalent message, and similarly, the second recycling order $(0,\, \frac{15}{24})$ describes the internal node labeled $\frac{9}{24}$ as the current message and m_1 as the closest neighbor node.

The encoder and decoder use two types of downscaling: **S2** and **S1**. The **S2** downscaling undoes E2 and **S1** undoes E1 upscaling. Let ρ be a variable that keeps the *recycled* bits, that need not to be sent to σ. The variable ρ is initialized to ϵ, where ϵ is the empty string. Each **S2** and **S1** yields `1` and `0` to ρ, respectively. Accordingly, from steps 9 to 11, **RecycleE** interprets the recycling orders $(1,\, \frac{3}{9})$ then $(0,\, \frac{15}{24})$ into the following sequence of downscaling: **S1**, **S1**, and **S2**, yielding the the corresponding binary sequence, `100`, to ρ. The **RecycleD** procedure is identical to **RecycleE** except that it inserts the bits that have been removed by the coder from σ, just to the left of σ, this entails to change only two lines of the programming of **RecycleE**, as described in the pseudo-code on lines 82–84.

In addition to the main function of the recycling procedure (accommodating the whole merged interval within G), let us explain why the coder proceeds in this certain way. If we look at W at step 11 and try to upscale the active interval of the message being encoded, $A = [a_1,\, a_2) = [8,\, 10)$, from right to left; i.e. upscale A in reverse order from step 11 to step 8, then the yielded bits that need to be sent to σ are the same as the bits kept in ρ! Which means that if Θ_{cur} is the proper thread for the next message, then the first three bits of the next message will be $\texttt{100} = \rho$, accordingly, these bits can be recycled, because it can be inferred implicitly by the decoder as described above.

We assumed that the next message to be encoded at time $t+1$ is m_3 without equivalents. The coder encodes m_3 as described above from steps 12 to 18. At step 18, procedure **Emit** is called to check if the current thread is to be continued (the proper thread of the message being encoded) or to be killed (not the proper thread). The thread is to be continued if the first bit of ρ (`100`) matches the first bit of w (**1**`000`) and, accordingly, the matched bits are removed (recycled) from ρ and w. (The first bit of a string w is extracted using $\text{head}(w)$ and the rest of w is extracted using $\text{tail}(w)$.) Otherwise, if no match is found, the current thread is then terminated since it will not be the proper thread to encode m_3 at time $t+1$. As a result, the encoder will send the compressed stream $\sigma = \texttt{10100}$ to the decoder, and the decoder can decode σ into the string $S = M_2 m_3$.

The pseudo-code, on lines 60 and 63, uses two types of rounding in procedures **S1** and **S2**: floor and ceiling. The floor and ceiling rounding operations are used to round down L and round up H, respectively. The reason behind using these two types of optimistic rounding is to ensure that there is no portion of the downscaled interval that will not be used (covered) by any concurrent thread, which would lead to the existence of a specific message that can not be encoded permanently.

The pseudo-code, on line 3, covers a special (highly skewed) case when the ratio of the received order occupies less than one tile or the entire available space ($\#I_{\text{cur}}$), to avoid assigning zero or one probability to any highly skewed event. Similarly, the pseudo-code, on line 54, handles another special (highly skewed) case for $\mathcal{M}_{\equiv}$, where the recycling procedures stop downscaling if $\#I_{\text{cur}}$ in A reaches the minimum, $\mathbb{B} = 1$, and allocates the maximum possible size to the highly probable message. Notice that the proposed technique was able to encode/decode $M_2 m_m$ and to improve the code efficiency using fixed-size (b-bit) registers, which is the main goal of FPACBR.

IV. Conclusion

In prior work, we have theoretically shown that ACBR can achieve a significant amount of better compression than HCBR. In this work, we have proposed FPACBR, which can be implemented and applied in practice. FPACBR is easy to interface with the compression techniques that suffer from the problem of redundancy caused by ME. As a future work, we intent to implement and apply FPACBR on the proper applications mentioned in this paper, in order to evaluate and measure its performance in practice comparably with the results obtained by HCBR.

References

[1] S. Mittal, "Lossless data compression for energy efficient transmission over wireless network," *Business Intelligence Journal*, vol. 5, no. 2, pp. 347–351, 2012.

[2] N. Kimura and S. Latifi, "A survey on data compression in wireless sensor networks," in *Proceedings of Information Technology Coding and Computing, 2005. ITCC 2005.*, April 2005, pp. 8–13.

[3] K. Barr and K. Asanovic, "Energy aware lossless data compression," in *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, May 2003.

[4] P. Volf and F. Willems, "Switching between two universal source algorithms," in *Proceedings of the IEEE Data Compression Conference*, March 1998, pp. 491–500.

[5] D. Knuth, "Efficient balanced codes," *IEEE Transactions on Information Theory*, vol. 32, no. 1, pp. 51–53, January 1986.

[6] D. Dubé and V. Beaudoin, "Recycling bits in LZ77-based compression," in *Proceedings of the Conférence des Sciences Électroniques, Technologies de l'Information et des Télécommunications*, Sousse, Tunisia, March 2005.

[7] ——, "Bit recycling with prefix codes," in *Proceedings of the Data Compression Conference*, Snowbird, Utah, USA, March 2007, p. 379.

[8] ——, "Improving LZ77 bit recycling using all matches," in *Proceedings of the IEEE International Symposium in Information Theory*, Toronto, Ontario, Canada, July 2008, pp. 985–989.

[9] A. Al-Rababa'a and D. Dubé, "Adaptation of bit recycling to arithmetic coding," in *Proceedings of the International Workshop on Systems, Signal Processing, and their Applications*, Algiers, Algeria, May 2013.

[10] ——, "Using bit recycling to reduce the redundancy in plurally parsable dictionaries," in *Proceedings of the 14th IEEE Canadian Workshop of Information Theory*, St. John's, Canada, 2015.

[11] S. Savari, "Variable-to-fixed length codes and plurally parsable dictionaries," in *Proceedings of Data Compression Conference, 1999. DCC'99*, March 1999, pp. 453–462.

Abbreviations

ME	Multiplicity of Encodings	ACBR	AC-based BR
HC	Huffman Coding	APAC	Arbitrary-Precision AC
AC	Arithmetic Coding	FPAC	Finite-Precision AC
BR	Bit Recycling	APACBR	Arbitrary-Precision ACBR
HCBR	HC-based BR	FPACBR	Finite-Precision ACBR
ND	Non-Determinism (or Non-Deterministic or Non-Deterministically)		